Storm: Program Reduction for Testing and Debugging Probabilistic Programming Systems

Saikat Dutta
University of Illinois, USA
saikatd2@illinois.edu

Wenxian Zhang
University of Illinois, USA
wzhan103@illinois.edu

Zixin Huang
University of Illinois, USA
zixinh2@illinois.edu

Sasa Misailovic
University of Illinois, USA
misailo@illinois.edu

ABSTRACT

Probabilistic programming languages offer an intuitive way to model uncertainty by representing complex probability models as simple probabilistic programs. Probabilistic programming systems (PP systems) hide the complexity of inference algorithms away from the program developer. Unfortunately, if a failure occurs during the run of a PP system, a developer typically has very little support in finding the part of the probabilistic program that causes the failure in the system.

This paper presents Storm, a novel general framework for reducing probabilistic programs. Given a probabilistic program (with associated data and inference arguments) that causes a failure in a PP system, Storm finds a smaller version of the program, data, and arguments that cause the same failure. Storm leverages both *generic* code and data transformations from compiler testing and *domain-specific*, probabilistic transformations. The paper presents new transformations that reduce the complexity of statements and expressions, reduce data size, and simplify inference arguments (e.g., the number of iterations of the inference algorithm).

We evaluated Storm on 47 programs that caused failures in two popular probabilistic programming systems, Stan and Pyro. Our experimental results show Storm's effectiveness. For Stan, our minimized programs have 49% less code, 67% less data, and 96% fewer iterations. For Pyro, our minimized programs have 58% less code, 96% less data, and 99% fewer iterations. We also show the benefits of Storm when debugging probabilistic programs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Probabilistic Programming Languages, Software Testing

ACM Reference Format:
Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: Program Reduction for Testing and Debugging Probabilistic Programming Systems. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338972

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338972

1 INTRODUCTION

Probabilistic programming languages offer an intuitive way to model uncertainty by representing complex probabilistic models as simple programs [1, 4, 7–9, 13, 16–18, 28, 32, 35, 42, 44, 52, 55]. A key novelty of probabilistic programming is the separation between the *probabilistic modeling* and *probabilistic inference*. An end-programmer expresses probabilistic models in a *high-level language* with constructs for random choice (e.g., sampling from common distributions), conditioning on data (e.g., observation statements) and probabilistic queries (e.g., posterior distribution) [19].

A *probabilistic programming system* (*PP system*) automates many intricate details of probabilistic inference, while executing one of the common inference algorithms, such as Monte-Carlo sampling [22, 39] or Variational inference [29]. A PP system takes three inputs: 1) a probabilistic program, 2) a set of data points on which to perform inference, and 3) arguments of the inference algorithm. Typically, PP systems compile the probabilistic program into an efficient low-level inference procedure, which includes initializing the underlying inference algorithm, translating of probabilistic programs (models) to an intermediate representation, simplifying the model, compiling to low-level API (e.g., Tensorflow), and many others.

The numerical and approximate nature of PP systems and implementation complexity make it hard to ensure their correctness, and subtle bugs can easily remain unnoticed [20, 38, 47]. Our recent study [10] showed that over 25% of all bugs in three popular systems are domain specific, including algorithmic, numerical, boundary condition, dimensional, and accuracy bugs. The bugs manifest as wrong results, crashes, infinite loops, or numerical exceptions.

If a failure occurs during the execution of a PP system, the developer typically has to figure out the source of the problem manually. This is not an easy task: while probabilistic programs are intuitive to write, they can be notoriously hard to analyze [26, 38]. Probabilistic programs typically have a small number of lines of code, but they exercise many functionalities of the underlying PP system. For instance, an execution of a simple 10-line program in Stan [15], one of the most popular PP systems, can execute over 6000 lines of code of Stan implementation.

To be able to reproduce and analyze the failures in PP systems, the developers of PP system suggest bug reports with self-contained and minimalistic tests, e.g., in Stan: *"the key to a successful bug report is to provide as much context as possible, ideally in the form of a small reproducible example"* [46]. This requirement is similar to the one from the standard compilers (e.g., [45]). Minimized programs help with both debugging (e.g., calculating the reference result) and speeding up regression testing (by executing the programs faster). However, coming up with minimal programs requires significant manual effort through trial-and-error.

Program reduction has been instrumental in the tasks of compiler testing and debugging. For standard compilers, researchers proposed various methods to reduce the size of bug-revealing programs such that the reduced program still exposes the bug, but helps the developer better understand and debug the program execution [36, 45, 51, 57]. Our insight is that the same conceptual approach can apply for debugging PP systems: just like a compiler that translates an input program, PP system translates and/or executes a triple of probabilistic program, data, and inference arguments. However, the existing program reduction approaches for conventional languages, either operate on textual representation [57], potentially generating many illegal programs, or use only syntactic information about the programs [36, 51], but do not leverage semantic information; they are also oblivious to the inference arguments (e.g., the number of Monte-Carlo samples).

Our Work. We present Storm, a novel approach for automatically reducing probabilistic programs and show its utility in the scenarios of testing and debugging PP systems. Storm applies various transformations to reduce the probabilistic programs. Unlike existing approaches for conventional programs, Storm leverages program analysis and probabilistic reasoning to simplify bug-revealing probabilistic programs. We show the benefits of the domain-specific and probabilistic information about the programs.

We formulate our problem in the spirit of [51]: Given a probabilistic program P, data d, and inference arguments θ that have a property ψ (e.g., a PP system execution fails with a particular error message), the goal of probabilistic program reduction is to find a smaller (P', d', θ'), that has the same property, $\psi(P', d', \theta') = \psi(P, d, \theta)$ (e.g., the PP system execution fails with the same error message).

Storm is a *generic framework* that uses both syntactic and domain-specific, semantic information about probabilistic programs to generate only valid probabilistic programs. We designed Storm to be language-agnostic: it translates programs from the existing systems to a common intermediate representation, Storm-IR. We define all our analyses and transformations on Storm-IR and finally output the reduced program back to the source language. We present the translation of two popular languages (Stan [15] and Pyro [4]) with significantly different syntax and language models.

Storm is a *transformation-based framework*. It supports both conventional program transformations and novel probabilistic transformations. Novel transformations include *data reducer* (which aims to keep specific bug-revealing values in the data set), *distribution simplifier* (which replaces complex distributions or parameters with appropriate constants, expressions, or simpler distributions), *parameter remover* (which removes a parameter and replaces its references with a suitable constant), *math-function call remover* (which replaces common mathematical functions with constants), and *inference argument reducer* (which finds a minimum number of samples or iterations of the inference algorithm required to reproduce a failure). They augment the basic program transformers, such as arithmetic simplifier, removers for conditionals, loops, function calls, and assignments (similar to C-Reduce [45]). Storm's reduction algorithm reduces the program size by iteratively applying both the basic and domain-specific transformations and performing lightweight analysis on the program's intermediate representation (including dependence, interval, type, and data-flow analysis).

Results. We use the reducer to generate smaller programs that reveal failures in two state-of-the-art PP systems: Stan [8, 15, 29], one of the most mature and frequently-used PP systems, and Pyro [4], a Python-based deep probabilistic modeling framework from Uber. We studied three sources of bugs: 1) a probabilistic bug database we created in previous research [10], which includes test programs that were already minimized by a human, 2) new bugs discovered using ProbFuzz [10], and 3) a repository with representative Stan models that offers larger probabilistic programs [48]. In total, we analyzed 47 programs (34 from Stan and 13 from Pyro).

Our results show that Storm's reduction strategies often generates significantly smaller programs than those provided by the users or developers. In particular, Storm was able to remove non-trivial program constructs in 45 programs, reduce the data size in 30 programs out of 33 programs that have data, and reduce the execution time of the inference algorithm (e.g., by reducing the number of Monte-Carlo simulations or Variational inference iterations) in 46 programs. Storm shows a significant improvement in the number of removed data points and program constructs over the baseline approach that applies only basic transformations.

Contributions. The paper makes the following contributions:

- ⋆ We present Storm, which is, to the best of our knowledge, the first reduction framework for probabilistic programs.
- ⋆ We introduce a program reduction algorithm that is aware of probabilistic information and guided by program analysis.
- ⋆ We introduce domain specific transformations in addition to basic transformations used for conventional languages.
- ⋆ We evaluate Storm on existing bug-revealing programs from popular probabilistic programming systems, Stan and Pyro.

2 EXAMPLE

Figure 1 presents a bug-revealing program in Stan, taken from the bug issue *Stan1610* [49]. We will demonstrate Storm's ability to reduce this program while still revealing the same bug in Stan.

2.1 Original Program

The program in Figure 1 represents a variant of Latent Dirichlet Allocation (LDA) model [5]. In LDA, each document is assumed to contain a mixture of topics and each topic is assumed to use a small set of words frequently. Using LDA model, users try to infer the distribution of words and topics in observed documents. The program in Figure 1 represents the topic distributions for users and items instead of documents. It consists of three parts:

- *Data block* (lines 1-13): It specifies the type and dimension of the input data which is to be used to condition the probabilistic model. It contains the dimensions and the names of all constants, priors, and observed data points.
- *Parameters block* (lines 14-18): It contains the random variables whose posterior distribution Stan should infer.
- *Model* (lines 19-36): The model establishes the relationship between the observed and unobserved variables. First, it assigns a *prior* to all the parameters, which denotes the user's belief of the distribution of their values. Here, all the parameters are assigned priors from *Dirichlet* distribution (lines 20-25). Then it specifies the relation of the variables to the data. It implements LDA using custom log-density updates (lines 27-33). The built-in

Dataset:

```
U = 28; I = 84; N = 326; K = 10; V = 17
word = [ (326 float values) ];
item = [ (326 float values) ];
user = [ (326 float values) ];
alpha_user = [ (10 float values) ];
alpha_item = [ (10 float values) ];
beta = [ (17 float values) ];
```

Model:

```
data {
  int K; // num topics
  int V; // num words
  int U; // num users
  int I; // num items
  int N; // total word instances
  int word[N]; // word n
  int item[N]; // item ID for word n
  int user[N]; // user ID for word n
  vector[K] alpha_user; // topic prior concentrations for users
  vector[K] alpha_item; // topic prior concentrations for items
  vector[V] beta; // prior probability for seeing word
}
parameters {
  simplex[K] item_topics[I]; // topic dist for item i
  simplex[K] user_topics[U]; // topic dist for user u
  simplex[V] word_topics[K]; // for topic k prob of seeing word v
}
model {
  for (i in 1:I)
    item_topics[i] ~ dirichlet(alpha_item); // prior on item topics
  for (u in 1:U)
    user_topics[u] ~ dirichlet(alpha_user); // prior on user topics
  for (k in 1:K)
    word_topics[k] ~ dirichlet(beta); // prior
  // for every word in our corpus
  for (n in 1:N) {
    real gamma[K];
    // for every topic
    for (k in 1:K){
    // topic distribution for this user
      gamma[k] <- log(item_topics[item[n], k] + user_topics[user[n], k])
            + log(word_topics[k, word[n]]);
    }
    increment_log_prob(log_sum_exp(gamma)); // likelihood
  }
}
```

Inference Arguments:

```
Engine = ADVI; Iters = 1000
```

Figure 1: Example – Original Code and Data

Dataset:

```
K = 1; V = 2
beta = [ 0.0588235, 0.0588235 ];
```

Model:

```
data {
  int K;
  int V;
  vector[V] beta;
}
parameters {
  simplex[V] word_topics[K];
}
model {
  for (k in 1:K)
    word_topics[k] ~ dirichlet(beta);
}
```

Inference Arguments:

```
Engine = ADVI; Iters = 125
```

Figure 2: Example: Reduced Code and Data

Reduction	%	Ratio
Lines of Code	69%	(11 / 36)
Code Constructs	83%	(12 / 70)
Data Points	98%	(41 B/4 KB)
Algorithm Iters.	87.5%	(125 / 1000)

Figure 3: Example: Reduction Statistics

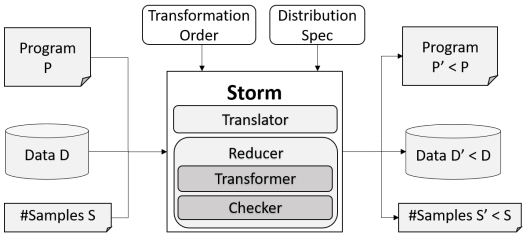

Figure 4: Storm Overview

function *increment_log_prob* updates explicitly the log density of the posterior distribution with the value of the inner expression.

Data. In addition to the program, the test case consists of data points, which give concrete values to all the constants and the vectors (the actual values omitted from Figure 1). For this program, the number of users `U` is 28, items `I` is 84, word instances `N` is 326, topics `K` is 10, unique words `V` is 17. Overall, the data size is around 4 KB.

Inference. The program runs with Stan's ADVI (variational) inference engine [29], which approximates the posterior distribution to a family of distributions with unknown parameters and converts the inference problem into an optimization problem. The algorithm then runs the model and tries to minimize the distance between the posterior and the chosen family of distributions for a given number of iterations. This number is given as the argument (1000).

Bug. This program produces `NaN` (Not-a-Number) in the output after 70 iterations when run using ADVI in Stan 2.7.0. The failure was due to a bug in the inference engine, which does not adapt its step-size sequence argument correctly, leading to `NaN`.

2.2 Reduced Program and Data

Figure 2 presents the test case minimized by Storm. The program now only samples from one Dirichlet distribution (instead of the previous complicated computation) and hence does not need to compute the posterior distribution. This program now has only 12 lines of code (compared to 36 in the original) and 12 instead

of the original 70 program constructs. Figure 3 presents the full reduction statistics. Because the model does not compute the posterior distributions, the new model does not need any of the user provided data (`word`, `item`, and `user`), which significantly simplifies reasoning about its correctness. Our manual inspection shows that the reduced program still reproduces the same bug as the original program, despite its much smaller size.

To reduce this program, Storm translates the source code to the intermediate language Storm-IR, transforms the program, and outputs the source of the reduced program. The transformations include basic (e.g., removing statements or expressions) and those specific to the probabilistic domain (e.g., Math-Function Call Remover, which replaces mathematical functions with appropriate constant values, and Data Reducer, which reduces data size). Storm applied basic transformations 17 times and domain-specific transformations 25 times to reduce the program. Eleven of these transformations were distinct. The results shows that Storm effectively leverages both basic and domain specific transformations.

2.3 Benefits of Program Reduction

Simplified Debugging and Fault-Localization: The reduced program immediately points out that the problem with this program may be caused by some interaction between the Dirichlet distribution and the inference engine. This is in contrast to the original program, where a developer would need to think about various aspects of the implementation – e.g., does the code correctly represent the model, how to simulate discrete distributions with log-probabilities – and the data – e.g., are the values and the parameters in range.

Easier to Derive the Reference Solution: The reduced program simply samples values from the Dirichlet distribution. Its probability density function can be easily obtained from a textbook:

$$p(x_1, x_2|beta1, beta2) = \frac{\Gamma(beta1 + beta2)}{\Gamma(beta1) \cdot \Gamma(beta2) \cdot x_1^{beta1-1} \cdot x_2^{beta2-1}}$$

By replacing *beta1* and *beta2* with values from Figure 2, the developer can compute the distribution of the program: $p(x_1, x_2) = 0.029 \cdot x_1^{-0.941} \cdot x_2^{-0.941}$. Then, the developer simply needs to check that the inference results conform to this probability distribution.

Reduction also helps for programs that do not have a closed-form solution. A common strategy is to use a different language or an inference language version and run Monte-Carlo simulation for a large number of iterations (e.g., over 10000 times) to get a good estimate of the distribution. Reduced program takes significantly less time to run than the original program, e.g., running Stan's NUTS engine for 10000 iterations on our reduced example takes 0.26 s, while running the original example takes 463 s (1781x slower).

Faster Regression Testing: Reducing the computation, data, and the iteration count directly translate to faster regression testing. Obtaining the reference solution also helps in creating effective regression test. Running the regression test for our example takes 0.02 s, while the original program is 214 times slower.

Other Applications of Program Reduction: Since it has the flexible choice of reduction objective ψ, Storm can be successfully used for other scenarios than reproducing bugs. We discuss one such case, minimizing the program while maintaining coverage in Section 7.

x	$\in$	*Vars*
c	$\in$	$Consts \cup \{-\infty, \infty\}$
aop	$\in$	$\{+, -, *, /, \hat{}\}$
bop	$\in$	$\{=, >, ...\}$
$Dist$	$\in$	{Normal, Uniform, Beta, ...}
ID	$\in$	*String*
Range	::=	<Expr ,Expr>
Dims	::=	[Expr^+]
Type	::=	`Int` | `Float` | Type Dims
Decl	::=	x : Type Limits? Dims? | x : [c^+]
Expr	::=	c | x Dims? | Expr *aop* Expr | Expr *bop* Expr | Function | *String*
Query	::=	`posterior`(x) | `expectation`(x)
Function	::=	ID(Expr^*)
FunctionDef	::=	`def` ID (($\text{Type } ID)^*$) { Statement^* }
Limits	::=	Range
Statement	::=	x = Expr | for $x \in$ Range; { Statement^* } | `observe`($Dist(\text{Expr}^+)$, x) | `if` (Expr) `then` Statement^* `else` Statement^* | x := $Dist(\text{Expr}^+)$ | Function | Decl
Program	::=	FunctionDef^* Statement^* Query^*

Figure 5: Syntax of Intermediate Representation

3 STORM OVERVIEW

Figure 4 presents the high-level overview of Storm. The inputs to Storm are 1) a probabilistic program, 2) data, and 3) the arguments of the inference (e.g., the number of samples). In each step, Storm checks whether the transformed (reduced) program satisfies the *reduction property* (ψ), a logical predicate that relates the outputs and the status of the original and the reduced programs. In this paper, we mainly consider the property that the reduced program reproduces the same error status and message as the original program.

3.1 Translators and Storm-IR

Storm translates each program to its intermediate representation, Storm-IR, on which it applies analysis and transformations. The translator is responsible for converting the program from the language of the existing PP system to the Storm intermediate representation *and* converting the reduced program and data from the intermediate representation back to the source language of the PP system. We developed translators for Stan and Pyro using Antlr [43].

Figure 5 presents the core syntax of Storm-IR. Storm-IR is an imperative language with standard constructs like arithmetic operations, *conditionals* and *loops*, and probabilistic constructs like distributions (*Dist*) and *observe* (which conditions the model based on given data). Each program in Storm-IR has three main components: user-defined functions (*FunctionDef*), a set of statements which describe the probabilistic model, and one or more probabilistic queries on the model. A query can be either be for the posterior distribution (*posterior*) of a parameter in the model or for its expected value (*expectation*). The Storm-IR has three kinds of variables: data variables, parameters, and local variables. The variables are declared as either primitives or n-dimensional arrays. Optionally, the parameters have

Algorithm 1: Storm Algorithm

Input: Program P_{in}, Data D_{in}, #Samples S_{in}, Transformation Order O
Output: Reduced Program P, Data D, #Samples S

```
procedure MINIMIZE
    Changed ← True
    P, D, S ← P_in, D_in, S_in
    while Changed do
        Changed ← False
        for T in O do
            P_red, D_red, S_red, C ← Transform(T, P, D, S)
            if C then
                P, D, S ← P_red, D_red, S_red
                Changed ← True
            end if
        end for
    end while
return P, D, S
end procedure
```

Algorithm 2: Transform Algorithm

Input: Transformation T, Program P_{in}, Data D_{in}, #Samples S_{in}
Output: Reduced Program P, Data D, #Samples S, Changed status C

```
procedure TRANSFORM
    C ← False
    P, D, S ← P_in, D_in, S_in
    L ← T.getLocations(P, D, S)
    i ← 0
    while i < L.length do
        P', D', S', Modified ← T.Transform(P, D, S, L(i))
        if Modified then
            Reproduced ← Checker(P', D', S')
            if Reproduced then
                P, D, S ← P', D', S'
                C ← True
                L.remove(i)
                i = 0
            else
                i = i + 1
            end if
        end if
    end while
return P, D, S, C
end procedure
```

a *Limits* construct which specifies the range in which the values of the parameter must be constrained during inference.

Stan and Pyro have significant differences in both syntax and the core design. For instance, Stan users need to specify the model in Stan's domain-specific language, which clearly separates the data, parameters, and the model code into different blocks. Pyro programs are written in Python, which makes it easier to write and compose different models. Unlike Stan, Pyro requires defining the posterior distribution for each parameter. Storm-IR is general enough to represent the core of majority of the example programs included in the repositories of these languages and allows the translators to handle the language-specific features discussed above. Our intermediate language draws inspiration from Probfuzz [10], but improves expressivity and generality (e.g., it allows arbitrary inter-leavings of statements like sampling, assignment, observes and loops). This allows the Storm-IR to represent a richer and more diverse set of probabilistic programs used across different PP systems.

3.2 Reduction Algorithm

Storm simplifies the structure of the programs by applying transformations and analyses on the intermediate representation. We describe the transformations we implemented in Section 4.

Helper Analyses. To ensure that the transformed program is syntactically correct, our transformers use several helper analyses for probabilistic programs. They include Dimensional, Type, Interval and standard DataFlow analyses (e.g. Def-Use). Dimensional analysis computes the dimension and type of any given expression. Interval analysis helps compute the range of values valid for a given expression. Def-Use analysis finds the uses of a variable in the model after it is declared. The transformations that simplify distribution expressions or replace parameters with constants can use the results of the analysis to make sure that the reduced program will not fail to run due to the range or dimension mismatches (e.g., it prevents setting the negative constant as the variance).

To support these analyses, Storm needs additional (domain-specific) information about common distributions and mathematical and probabilistic functions. The information includes the names and ranges of parameters and their support (the ranges of their outputs). For instance, the specification of Normal distribution states that the distribution is continuous and has unconstrained support; its first parameter (the mean) is an unbounded real and the second one (the variance) is a positive real.

Main Algorithm. Algorithm 1 presents the reduction algorithm. It takes the program P, data D, and number of samples or iterations S (if available). It can optionally take the order of transformations O which is to be used during reduction process. The algorithm is iterative fixed point computation, which in each step tries to apply the transformations and then checks whether the reduced programs satisfy the reduction property. The variable *Changed* tracks whether any transformation was successfully applied during the current iteration. In each iteration, the algorithm tries to reduce the program using each transformer T according to the pre-specified order O. The algorithm stops when the iteration cannot apply any reduction.

The *Transform* algorithm (Algorithm 2) takes as input a transformer T, program P_{in}, data D_{in}, and number of samples S_{in}. It finds all candidate locations for the transformation in the program for transformer T using *getLocations* function. For transformations which reduce data, this would return the data items in the program. For Inference Argument Reducer, this returns the inference parameters to reduce (in this case only Samples S). Next, for each candidate, the transformer T tries to transform it and check if the reduced program still reproduces the same failure as the non-reduced version using the *Checker* function. If it succeeds, then the triple of program, data, and, samples are updated, the candidate is removed from the list L and i is reset to 0. Otherwise, it moves to the next candidate. Resetting i to zero allows the transformer to re-check the previous locations which may now be modifiable after the recent change. Finally, the algorithm returns the program, data, and samples along with the indicator of whether any of them were transformed.

The *Checker* translates the program in Storm-IR back to the source code, runs it on the data, and monitors the execution. For program failures, it is often sufficient to expect the *exact* exception strings in output, e.g. *"Input vector ... is -nan"* or *"error: invalid cast from type 'stan::math::var' to type 'double'"*. For infinite loops, we set a reasonable timeout interval; the transformed program reproduces the failure if it still times out (like the original program). The Checker returns "True" if the execution of the candidate program still has the property of interest (e.g., the same error message), or returns "False" if the transformed program does not have this property.

4 TRANSFORMATIONS

We divide Storm's transformations into *basic* – typical structural reduction transformations that do not require probabilistic domain-knowledge – and *probabilistic* – that use the domain-knowledge. We describe the transformations next, and formally specify them in the Appendix [50].

4.1 Basic Transformations

Storm implements the common statement-level and expression-level transformations, which can be found in conventional program reduction tools, such as C-Reduce [45]. The transformations include *Loop Remover* (removes entire loop), *Loop Variable Remover* (the loop variable is replaced with a constant), *Conditional Remover* (randomly chooses one of the two branches), *Function Statement Remover* (removes a function call statement), and *Dead Variable Remover* (finds variables and data items which have been assigned constant values or sampled from distributions but never used).

Storm also has an *Arithmetic Simplifier*, which reduces arithmetic expressions by replacing variables with constants, complex expressions with simpler, etc. For instance, it can convert $a + b$ to either a or b. Since the operands can be arrays or vectors or matrices, Storm performs type and dimensional analysis of the expression and replaces the expression with an appropriate constant-valued data structure. To reduce non-determinism, the arithmetic simplifier always tries to remove the first operand first. Only if the reduced program fails to reproduce the failure, it tries to remove the second operand and checks for failure reproduction again.

4.2 Probabilistic Transformations

Data Reducer. The input data in Storm-IR contains primitives like integers or floating-point numbers or more complex data structures like vectors and matrices. In some cases, boundary or special values in the data may cause run-time failures. Isolating these values in a smaller data set can ease debugging.

The Data Reducer picks one data item (typically a vector or a matrix) and tries to reduce it, by successively subdividing the number of values that remain in the data structure. This transformation also checks that any related data items maintain the same dimensions. For example, in a linear regression model, which models $y = a \cdot x + b$, it is important that the arrays with values for x and y have the same size, and therefore Storm reduces them in the same way and with the corresponding data values.

Parameter Remover. The unobserved variables which must be inferred from the observed data are usually specified as parameters in Storm-IR. This transformer replaces the use of the parameter with a constant value, vector, or matrix. Storm chooses the constant values randomly, from the set of those within the support of the prior distribution and ensures that the dimensions are maintained.

Math-Function Call Remover. Stan and Pyro provide inbuilt mathematical functions (*log*, *exp*, *abs*, etc.) and probability-related functions (*logit*, *tgamma*, *gamma_p*, etc.). Storm replaces such function calls with a value in the expected output range of the function. These kinds of functions require domain knowledge; for example, the output of *gamma_p* is always positive. For overloaded functions, Storm performs type and dimensional analyses on the function arguments to ensure that the expression is valid (such analyses are typically not done by the conventional program reducing approaches).

Distribution Simplifier. This transformation replaces less-often used distributions like *Laplace*, *Weibull*, etc. with more commonly used distributions like *Normal* or *Uniform*. If the program already uses the simpler distributions, it tries to reduce the parameters to standard values. For example, it might reduce $Normal(52.15, 10.2)$ to $Normal(0, 1)$, a standard normal distribution. These transformations are useful when a developer wants to understand the reduced program and manually calculate reference solutions. They can also help with fault-localization, by pinpointing that the failure is (not) due to less-commonly used distributions.

Inference Argument Reducer. Stan and Pyro implement two kinds of Monte-Carlo sampling algorithms: HMC [39] and NUTS [22]. They allow the user to specify the number of iterations to run, which determines the number of samples to be taken from the posterior distribution for inference. For variational inference algorithms, like ADVI [29] in Stan and SVI [27] in Pyro, the iterations determine the maximum steps the optimization algorithm might use. The Inference Argument Reducer searches for the minimum number of iterations that reproduces the failure. In each round, the Inference Argument Reducer halves the iterations, starting from the initial value, and checks whether the failure is reproduced.

Limits Remover. For every data and parameter variable, some languages (Stan being a prominent example) allow the user to specify a set of lower and/or upper limits of the parameters. The limiting helps the sampling algorithm focus on a subset of the input space and converge faster. But these limits can be ill-specified in practice. This transformer attempts to remove the limits and checks if the program still fails. It is analogous to changing the variable type in conventional programs.

4.3 Transformation Orders

An important component of Algorithm 1 is the transformation order O, which can affect the quality and speed of reduction on a given benchmark. We evaluated our algorithm on six orders, which we briefly outline here and specify them fully in the Appendix [50].

Random order (**Rnd**) chooses each transformation with uniform probability without replacement. Fixed order (**Fixed**) is an order we manually chose for the experiments based on our experience and understanding of transformations. Size of transformation order (**Size**) applies first the transformations that change more code (e.g., *Loop Remover* and *Conditional Remover*). Cost of analysis order (**CoA**) ranks the transformations such that transformations which do not require any analyses (e.g., *LoopRemover*, and *FunctionStatementRemover*) execute before the transformations which require

one or more analyses techniques (e.g., *Arithmetic Simplifier*). Basic-Probabilistic order (**B-P**) applies first all basic transformations, then probabilistic ones. Probabilistic-Basic order (**P-B**) first applies all probabilistic transformations, than basic ones.

5 METHODOLOGY

This section presents our methodology for collecting and categorizing program that expose bugs in Stan and Pyro.

5.1 Selection of Bugs

Stan. To obtain probabilistic programs that reveal existing bugs for Stan, we studied the bug reports from the existing bug-database for probabilistic programming systems [10]. The database contains 138 probabilistic bugs, divided in four categories. We denote each such bug with the prefix "stan" followed by the issue identifier. We selected only the bugs with reproducible test cases.

We augment the programs from the bug reports with additional bug-revealing programs from Stan's repository of models [48]. We obtained a set of 367 probabilistic programs from Stan's public repository [48]. We ran the programs across the versions of Stan 2.3, 2.5, 2.6, 2.6.2, 2.7, 2.9, 2.10, 2.14, 2.15 and 2.18 (the latest). We ran each program using three inference algorithms available in Stan: NUTS [22], HMC [39], and ADVI [29]. We identified those programs that produce a failure (compile but either crash, produce numerical errors, or loop infinitely) in one of the earlier versions, but produce the correct result in the latest version. Those programs are representative of those that would reveal true bugs in the PP system in real operation. These programs were considerably larger than the ones obtained from the bug reports in both code (more than 90%) and data (more than 100%) on average.

In total, we tested Storm on 23 bugs from Stan issues and 11 programs taken from Stan's example models repository. The size of test cases range from 5 to 57 lines of code (excluding blank lines and comments). We used Cmdstan to run all the programs, except three that require PyStan. Overall, the programs cover four inference methods: Sampling (NUTS [22] and HMC [39]), Variational (ADVI [29]) and Optimization (also known as MAP [41]), and one simulation method: Fixed Param (FP); additionally, some failures were due to bugs in the Stan compiler code (stan723), and bugs in Diagnostic mode (stan1308), which is used to test computations of gradient and log-probability and flag any issues.

Pyro. For Pyro, we collected bug-revealing programs from two sources. We obtained 6 bug-revealing programs from [10]. We named those programs pf1-pf6. We also converted the programs obtained from Stan's repository of models [48] into Pyro programs and ran them using the three recent versions of Pyro, 0.2.0, 0.2.1, and 0.3.0. We identified 7 programs (dyes, dyes_020, ES, ES_020, GP2, GP2_020, radon) which run without failures in the current released version of Pyro (0.3.0), but crash in the older and buggy versions. The program lines range from 36 to 77. All the programs were run using Stochastic Variational Inference (SVI) algorithm.

5.2 Bug Classification

Following the characterization from [10], we classified the bugs as:

Crashing Bugs. These bugs cause compilation-time or run-time failures with error messages in the output, such as *"runtime error: load of value 3, which is not a valid value for type 'bool'"*, *"Segmentation fault: 11"*, *"Domain error in arguments"* etc. Many of these bugs are due to the out-of-bounds accesses or wrong dimensions of the data structures. We reproduced 5 such bugs in this category for Stan and all 13 Pyro bugs fall in this category.

Numerical Bugs/Infinite Loops. Numerical bugs include special values like `NaN`s or `Inf`s in the output, which usually appear due to missing support for handling boundary conditions. We reproduced 9 Stan bugs from this category with inputs provided by the bug reports. A special class of numerical bugs are those that cause infinite loops during inference. We reproduced 2 Stan infinite loop bugs.

Accuracy/Unexpected Output Issues. For the cases, the execution does not crash but produces some unexpected values. For example, in one case, the computation of effective sample size of a parameter for the NUTS engine in Stan was incorrect due to a bug in the code. We reproduced 5 bugs from this category with inputs provided by Stan's bug reports.

Language/Implementation. These bugs appear while translating the program's source-code. We replicated 8 such bugs for Stan. Finally, we also consider 3 general coding bugs.

5.3 Reduction Metrics

To demonstrate the quality of test case minimization, we collect several metrics during experimentation. For code reduction, we consider two metrics that characterize the size of the program:

- Lines of code, without empty lines or comments.
- The count of non-terminal language constructs in Stan's parse tree (e.g., loops, sampling statements, conditionals, or arithmetic operations) for the given test case.

We use count of non-terminals since the probabilistic programs have a high-level of expressiveness, and a single change (smaller than a line) in the code may make significant impact on the accuracy, analyzability, or execution time of the program. For the grammar that we used for Stan, there are 42 such unique constructs. For Pyro, we use the Python grammar, which has 41 unique constructs.

In this work, we use a metric for code reduction known as *Size Reduction Rate (SRR)* defined in [2] as:

$$Score(t_o, t_{red}) = \frac{Size(t_o) - Size(t_{red})}{Size(t_o)},$$

where t_o is the original test case, t_{red} is the reduced test case, $Size(t)$ is the size of the test case using the metrics defined above. To compare data savings *DataRed*, we use the following metric:

$$DataRed(t_o, t_{red}) = \frac{DataSize(t_o) - DataSize(t_{red})}{DataSize(t_o)}$$

where $DataSize(t)$ is number of bytes in the data input for t. Finally, we calculate the ratio of the number of samples/iterations in the reduced test case to that in the original.

6 EVALUATION

We evaluate experimentally the following research questions:

RQ1 How effective is Storm in reducing test cases?
RQ2 How much benefit do probabilistic transformations provide?
RQ3 How much does program reduction speed up inference?
RQ4 How important is the order of transformation in reduction?

Table 1: Stan Example-Models Reduced Using Storm

Test	SRR	LoC	Data Red.	Iters.
arma11 (VI)	44/85	20/25	49.9%	3/1000
arma11_alt (VI)	46/73	20/23	49.9%	1/1000
dogs_log (VI)	10/98	6/33	100.0%	1/1000
roaches (VI)	11/38	8/18	99.5%	1/1000
roaches_od (VI)	23/74	12/28	99.2%	1/1000
roaches_od_2 (VI)	23/74	14/28	64.8%	1/1000
salm2 (VI)	24/74	16/27	51.1%	1/1000
salm2_2 (VI)	1/74	2/27	100.0%	1/1000
salm (VI)	38/128	22/47	41.0%	1000/1000
stagnant (VI)	11/76	9/26	98.7%	125/1000
survey (VI)	35/65	28/32	36.4%	1/1000
Avg. Savings	68.37%	49.27%	71.86%	103/1000

Table 2: Stan Github-Issues Reduced Using Storm

Test	SRR	LoC	Data Red.	Iters.
stan240 (NUTS)	8/8	9/9	0.0%	1/1000
stan499 (NUTS)	12/17	12/14	68.2%	1/1000
stan543 (NUTS)	18/32	14/18	74.5%	500/1000
stan674 (NUTS)	13/24	8/10	NA	1/1000
stan685 (NUTS)	8/13	6/10	NA	1/1000
stan723 (NUTS)	23/32	16/19	NA	1/1000
stan1053 (FP)	13/21	8/11	NA	1/10000
stan1121 (NUTS)	5/16	8/13	0.0%	1/1000
stan1194 (NUTS)	5/5	5/5	NA	1/1000
stan1200 (Opt)	12/21	11/15	99.7%	1/1000
stan1241 (NUTS)	19/28	8/13	NA	1/1000
stan1308 (Diag)	37/196	11/57	100.0%	1/1000
stan1366 (NUTS)	12/16	6/11	NA	1/1000
stan1435 (Opt)	5/20	8/13	0.0%	1/1000
stan1443 (NUTS)	8/13	7/10	NA	1/1000
stan1474 (NUTS)	14/16	10/10	NA	15/1000
stan1610 (VI)	12/70	11/36	98.8%	125/1000
stan1789 (NUTS)	10/23	9/16	NA	1/1000
stan1974 (NUTS)	5/7	3/6	NA	1/1000
stan2188 (NUTS)	9/9	6/6	NA	1/1000
stan2237 (HMC)	3/63	5/25	100.0%	1/1000
stan2294 (NUTS)	9/9	6/6	NA	1/1000
stan2311 (NUTS)	4/18	6/16	NA	1/1000
Avg. Savings	40.89%	32.32%	60.13%	29/1391

Table 3: Pyro Example-Models Reduced Using Storm

Test	SRR	LoC	Data Red.	Iters.
pf_1 (VI)	138/209	28/37	99.3%	3/4000
pf_2 (VI)	88/207	23/36	99.3%	1/4000
pf_3 (VI)	153/298	30/46	98.6%	3/4000
pf_4 (VI)	153/285	30/44	92.2%	3/4000
pf_5 (VI)	126/270	27/44	98.8%	62/4000
pf_6 (VI)	153/326	30/50	92.1%	3/4000
dyes (VI)	131/331	29/55	96.3%	1/4000
dyes_020 (VI)	129/330	29/56	96.3%	1/4000
ES (VI)	129/250	29/44	95.8%	1/4000
ES_020 (VI)	129/254	29/44	95.8%	1/4000
GP2 (VI)	131/553	29/77	99.9%	1/4000
GP2_020 (VI)	129/557	29/77	99.9%	1/4000
radon (VI)	129/439	29/61	100.0%	1/4000
Avg. Savings	56.58%	42.04%	97.25%	6/4000

6.1 Test Cases Reduced by Storm

Tables 1, 2, and 3 show the performance of Storm for the bugs in Stan examples, Stan issues, and Pyro examples, respectively. Column 1 (**Test**) is the test-case identifier – an issue number or benchmark name for the example-models, and algorithm[NUTS/ HMC/ Variational(VI)/Optimize(Opt)] or mode[Diagnose(Diag)/Fixed-Param(FP)] used to run the program. Column 2 (**SRR**) presents the ratio of the original number of program constructs to the ones in reduced test cases (Section 5.3). Column 3 (**LoC**) presents the ratio of original source lines of code to the reduced source lines of code for the test case. Column 4 (**Data Red.**) presents the percentage of the reduced data points (relative to the original size). The cases which did not have any data are marked as NA. Column 5 (**Iters**) presents the reduction of the argument (the number of samples for MCMC, iterations for variational inference) of the approximate inference algorithms. For **SRR**, **LoC**, and, **Data Red.**, we compute the average savings by adding up the savings for each benchmark and dividing by total benchmarks in the set. For **Iters**, we compute the average of original iterations and reduced iterations separately and report the ratio as savings.

From the data in the three tables, we conclude that Storm was able to reduce all 47 test cases across all algorithms in at least one of program constructs, number of lines, data, or the number of samples. Storm was able to reduce over 90% in program size, data, and inference arguments (samples in Monte-Carlo simulation and iterations in Variational inference). Out of total 47 programs, 5 improved in one category, 7 improved in two categories and 29 improved in three categories. Tables 1 and 3 show that with an exception of salm, all larger probabilistic programs are reduced by Storm in all three categories.

Coverage. Table 4 presents how many lines of the PP system the original and reduced program cover on average (as measured with gcov for Stan and coverage.py for Pyro). Column 1 (**Benchmark**) presents the group of benchmarks. Column 2 (**Hit**) presents the average number of lines executed by the original programs. Column 3 (**Total**) presents the total number of lines in the PP system. Column 4 (**Cov**) presents the average original line coverage for the benchmarks. Column 5 (**HitR**) presents the average number of lines executed by the reduced programs. Column 6 (**TotalR**) presents the total lines in the PP system. Column 7 (**CovR**) presents the average line coverage for the reduced programs.

Table 4: Coverage of Reduced Programs

Benchmarks	Hit	Total	Cov	HitR	TotalR	CovR
Stan Issues	9796	25713	38.07%	8996	25713	34.97%
Stan Examples	10690	25738	41.53%	9844	25738	38.24%
Pyro Examples	7272	24790	29.31%	7224	24790	28.92%

The results show that 1) the number of lines covered by both the original and the reduced programs is significant and 2) the coverage of the reduced programs, despite of their significantly smaller size is only slightly lower than the coverage of the original programs.

6.2 Impact of Probabilistic Transformations

We next study the impact of the newly proposed probabilistic transformations. To do so, we compare the impact of Storm when using both the probabilistic and basic transformations, to a variant that

uses only basic transformations. Table 5 presents the summary of the results. For each group of benchmarks, we compute and aggregate three statistics from Section 6.1. We compared the savings of that version of Storm to the original (non-reduced) program. Here, Column 2 (**Code**) presents savings in code constructs – SRR. Column 3 (**Data**) presents savings in data items. Column 4 (**Iters**) presents savings in the number of iterations. Note that the basic transformations cannot not reduce the number of iterations. In all cases, the savings are represented as percentages.

Table 5: Comparing Storm and Basic Transformations Only

Benchmarks	Code		Data		Iters	
	Storm	Basic	Storm	Basic	Storm	Basic
Stan Issues	40%	36%	60%	46%	97%	0%
Stan Examples	68%	61%	71%	47%	89%	0%
Pyro Examples	56%	53%	97%	82%	99%	0%

The results show that probabilistic transformations improve reduction of both code and data. The reduction in data with basic transformations is due to the *Dead Variable Remover* transformer: when it is possible to remove some data variables from the model, corresponding data-sets can also be removed from the data file without any effect on the model. Even then, we notice that in three cases, probabilistic transformations can further reduce the data-sets.

Probabilistic transformations contributed significantly to the success of reduction – more than 60% of the successful transformations across the three sets of benchmarks were domain-specific transformations. The Storm algorithm accepted 59% of all the domain-specific transformations, compared to 48% of the basic transformations across all benchmarks.

The reduction of inference arguments is unique to Storm with probabilistic transformations. For all the test cases which use sampling algorithms, Storm was able to significantly reduce the number of samples (to 1 in all cases except stan543 and stan1474). For variational inference, the iterations were reduced to 1 in 8 cases (out of 12) for Stan and 8 cases (out of 13) for Pyro. This shows that often the bugs can be revealed quickly by a small number of iterations and can save debugging time for the developer.

6.3 Speedup of Reduced Programs

If the developers needs to rerun these tests in regression testing, they can leverage smaller versions of the programs provided by Storm's transformations. Table 6 presents the summary of the runtimes (without compilation) of the original and reduced programs, run with recent versions of Stan (2.16.0) and Pyro (0.2.1). We only consider the cases where both original and reduced programs pass the test. Column 2 (**TimeO**) presents the average time of the original program. Column 3 (**TimeR**) presents the average time of the reduced program. Column 4 (**Speedup**) presents the ratio of TimeO by TimeR. Overall, the speedup when running the reduced tests is significant, especially for the larger programs with more data (for Stan and Pyro examples).

Table 6: Execution time reduction

Benchmarks	TimeO	TimeR	Speedup
Stan Issues	1.02s	0.25s	4.1x
Stan Examples	12.63s	0.10s	126.3x
Pyro Examples	106.36s	0.85s	125.1x

6.4 Impact of Transformation Orders

We evaluated whether the order of transformations in each step of the algorithm has a significant impact on the overall reduction of the test cases. We ran Storm on the test cases using six orders described in Section 4.3.

Table 7 presents the total execution times of the algorithm for these six orders, on a 12-core machine, using all cores for evaluation. Each time is in the format "minutes:seconds". In all cases, Storm converges to the minimal test case, or a test case with very similar quantitative reduction metrics (which we omitted), even in unfavorable orderings. Recall that Storm's algorithm iterates until reaching a fixed point, and unfavorable ordering will most often simply take more steps to terminate.

Table 7: Execution Times for Different Orders.

	Fixed	Rnd	Size	CoA	B-P	P-B
Stan Issues	41:24	45:04	53:33	41:32	55:58	44:27
Stan Examples	47:33	43:26	42:15	46:15	46:32	52:28
Pyro Examples	2:25	7:14	4:52	1:46	2:05	2:18

For Stan Issues, Fixed and CoA orders show the best results. For Stan Examples, Size order is the fastest. We believe the reason is that these programs are more complex than the Stan issues and have more control structures like loops and conditions. Hence, using the transformations which remove the larger blocks early on reduces the run-time of the programs and thus the reduction algorithm as well. Finally, for Pyro examples, CoA is the fastest. The time of algorithm for Pyro is significantly smaller than for Stan because it interprets the programs, instead of compiling them, like Stan.

7 APPLICATIONS OF STORM

In this section, we highlight two additional scenarios in which Storm can be applicable.

7.1 Incremental Debugging

A test case can potentially reveal multiple bugs in the system. But during execution, one failure can hide other bugs. A developer then has to go through a cycle of fixing a bug and re-running the test case to find other bugs in the system. Storm can help automate this cumbersome process.

Consider a simplified program for linear regression in Figure 6. The program has two data-sets x and y, each of size 10 (lines 2 and 3). In lines 4-6, the parameters w, b, and p, are assigned prior distributions. In line 7, the linear regression model is defined and conditioned on the data variables using observe statement. Lines 8-10 contain queries for posterior distribution for each parameter. The original program fails when run with Pyro 0.1.2 with the error *"Domain error in arguments"*, which does not clearly indicate the cause of the failure. When we run this program with Storm, it is reduced to the program from Figure 7. The data-set y now has only 1 element and the observe statement has a simple distribution and data variable y. Now, it is quite easier to figure out that the value in y (-2.99) is outside the range of support for beta distribution $(0, 1)$. If we look at the original program, we can observe that the values in y were outside of the support of lognormal distribution, $(0, \infty)$.

Even after this issue is resolved (for instance, by changing *log-normal* distribution to *normal*), the program fails with the same

```
1  N: 10
2  x: [ 72.97, 34.94, ...]
3  y: [ −2.99, 1.95, 2.77, ...]
4  w:= exponential(37.47)
5  b:= exponential(−31.49)[N]
6  p:= lognormal(55.43,61.35)[N]
7  observe(lognormal(w*x+b, p), y)
8  posterior(w)
9  posterior(b)
10 posterior(p)
```

Figure 6: Original Program

```
1  N:1
2  y: [−2.99]
3  p:= gamma(1.0,1.0)[N]
4  observe(beta(1.0, 1.0), y)
5  posterior(p)
```

Figure 7: Minimized Prog. 1

```
1  N:3
2  b:= exponential(−31.49)[N]
3  posterior(b)
```

Figure 8: Minimized Prog. 2

error. Using Storm again, we reduce the fixed original program (Figure 6) to the snippet in Figure 8, which has only one parameter with exponential distribution. The bug here is the negative value in exponential distribution, which expects a positive value. The original program had the same issue at Line 5. Storm takes only 74 seconds to find the error-inducing lines in each round.

7.2 Using Coverage as a Criterion

We explore the generality of Storm through a case study where we change the reduction property (ψ) to preserve the line coverage of the PP system (Stan) under test i.e. the coverage of the reduced program is the same as the original program. We used *lcov* to measure the line coverage after each transformation. We also turned off transformations that do not always reduce code and might cause the transformed program to execute a different function (e.g., Distribution Simplifier may replace a distribution with another distribution). Overall, Storm reduced the code constructs by 30.4% and data by 22.5%.

8 THREATS TO VALIDITY

Internal. Our Storm implementation may contain bugs, some bugs may have been mis-categorized during our selection and reproduction, and we may have made wrong conclusions about some minimized programs. To mitigate the risk of implementation bugs, multiple co-authors conducted a code-review of Storm and test-cases.
External. Storm methodology may not generalize to all PP system. However, there are three aspects which help mitigate the risk. First, we present the evaluation on two commonly used languages with different design, Stan and Pyro. Second, we looked at historical bugs, which may provide a good guide for the kind of bugs that may appear in the future. The maturity of Stan and the development effort in Pyro increase confidence that these bugs are representative of probabilistic programming in general. Third, our design on Storm minimizes the dependence on the language – most of the transformations are generic and can be applied to other languages.

9 RELATED WORK

Test Reduction. C-Reduce [45] reduces test cases for C programs (often generated using CSmith [56] test generator). C-Reduce uses source-to-source transformations customized for C-like programs, but its application to the domain of probabilistic programming is not straightforward. We also show the importance of domain-specific (probabilistic) transformations for successful program reduction, and make a parallel to CSmith and C-Reduce, by showing how Storm can reduce the programs generated by Probfuzz [10]. Other approaches for program reduction include Delta debugging [57], which is one of the earliest known techniques for test reduction. It removes parts of the failing test (code or data) until no single part can be removed without the test passing. Hierarchical Delta Debugging (HDD) [36] applies DD using the structure of the input. HDD generates fewer syntactically invalid programs but provides no guarantee. In contrast to these approaches, Storm produces only syntactically-correct reduced tests.

Perses [51] is a recent language-agnostic framework for reducing programs in conventional programming languages (e.g., C and Java). Like Perses, Storm uses the syntax of the language to guide the reduction process, applies the transformations on the intermediate representation, and generates syntactically valid programs. Storm strengthens the reduction process using various kinds of static analyses including dimensional and type analysis. Storm also uses probabilistic transformations to reduce data and inference parameters, which improves the overall reduction quality beyond the reach of general reduction frameworks.

Zhang et al. [58] proposed a technique for test simplification that is also able to modify portions of a test by replacing expressions with those already existing in the test. Other approaches [21, 30, 53] provide domain specific transformations to produce minimal structured data sets and reduce size of tests. To the best of our knowledge, we are the first to present domain-specific transformations and minimization for the probabilistic programming domain.

Verification and Analysis of Probabilistic Programs. Previous research proposed various techniques for statically analyzing and verifying properties of probabilistic programs, e.g., [6, 11, 14, 25, 31, 33, 37], including analyses that aim to help with debugging, e.g., [23, 38, 40]. In comparison, Storm presents a dynamic analysis that performs a heuristic search for smaller probabilistic program that reveal the same bugs.

Program slicing [54] is a standard technique for removing unnecessary parts of the code. Researchers recently extended slicing to probabilistic setting [3, 24]. Similarly, refactoring is a general set of techniques that modify program source code while preserving the program semantics, yet improve the program's internal structure [12, 34]. In contrast, Storm reduces program while ensuring that only bug manifestation remains, and has the freedom to change program semantics.

10 CONCLUSION

This paper presented Storm, a novel approach for reducing probabilistic programs. Storm presented both basic program reduction transformations, driven by program analysis and domain-specific probabilistic techniques to minimize the size and complexity of bug-revealing probabilistic programs. We evaluated Storm on 47 bug-revealing programs from two state-of-the-art PP systems, Stan and Pyro. For Stan, our minimized programs have 49% less code, 67% less data, and 96% fewer iterations. For Pyro, our minimized programs have 58% less code, 96% less data, and 99% fewer iterations.

ACKNOWLEDGEMENTS

This work was funded in part by NSF (Grants No. CCF-1629431, CCF-1703637, and CCF-1846354) and DARPA. We would also like to thank Owolabi Legunsen, and the anonymous reviewers for their comments on the research presented in this paper.

REFERENCES

[1] J. Ai, N. S. Arora, T. Jiang, M. Tingley, et al. 2019. HackPPL: a universal probabilistic programming language. In *MAPL*.

[2] M. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce. 2016. Evaluating non-adequate test-case reduction. In *ASE*.

[3] T. Amtoft and A. Banerjee. 2016. A theory of slicing for probabilistic control flow graphs. In *FoSSaCS*.

[4] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* (2018).

[5] D. Blei, A. Ng, and M. Jordan. 2003. Latent Dirichlet Allocation. *Journal of Machine Learning Research* (2003).

[6] M. Borges, A. Filieri, M. d'Amorim, and C. Păsăreanu. 2015. Iterative distribution-aware sampling for probabilistic symbolic execution. In *FSE*.

[7] J. Borgström, A. Gordon, M. Greenberg, J. Margetson, and J. Van Gael. 2011. Measure transformer semantics for Bayesian machine learning. In *ESOP*.

[8] B. Carpenter, A. Gelman, M. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, A. Riddell, et al. 2016. Stan: A probabilistic programming language. *JSTATSOFT* (2016).

[9] G. Claret, S. Rajamani, A. Nori, A. Gordon, and J. Borgström. 2013. Bayesian inference using data flow analysis. In *FSE*.

[10] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic. 2018. Testing probabilistic programming systems. In *FSE*.

[11] A. Filieri, C. Păsăreanu, and W. Visser. 2013. Reliability analysis in symbolic pathfinder. In *ICSE*.

[12] M. Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[13] T. Gehr, S. Misailovic, and M. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV*.

[14] J. Geldenhuys, M. Dwyer, and W. Visser. 2012. Probabilistic symbolic execution. In *ISSTA*.

[15] A. Gelman, D. Lee, and J. Guo. 2015. Stan A Probabilistic Programming Language for Bayesian Inference and Optimization. *Journal of Educational and Behavioural Statistics* (2015).

[16] W. Gilks, A. Thomas, and D. Spiegelhalter. 1994. A language and program for complex Bayesian modelling. *The Statistician* (1994).

[17] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. 2008. Church: A language for generative models. In *UAI*.

[18] N. Goodman and A. Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org.

[19] A. Gordon, T. Henzinger, A. Nori, and S. Rajamani. 2014. Probabilistic programming. In *FOSE*.

[20] R. Grosse, S. Ancha, and D. Roy. 2016. Measuring the reliability of MCMC inference with bidirectional Monte Carlo. In *NIPS*.

[21] S. Herfert, J. Patra, and M. Pradel. 2017. Automatically reducing tree-structured test inputs. In *ASE*.

[22] M. Hoffman and A. Gelman. 2014. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* (2014).

[23] Z. Huang, Z. Wang, and S. Misailovic. 2018. Psense: Automatic sensitivity analysis for probabilistic programs. In *ATVA*.

[24] C. Hur, A. Nori, S. Rajamani, and S. Samuel. 2014. Slicing probabilistic programs. In *PLDI*.

[25] N. Jansen, C. Dehnert, B. L. Kaminski, J. Katoen, and L. Westhofen. 2016. Bounded Model Checking for Probabilistic Programs. *ATVA*.

[26] J. Katoen. 2015. Probabilistic Programming: A True Verification Challenge. In *ATVA*.

[27] D. P. Kingma and M. Welling. 2014. Auto-Encoding Variational Bayes. In *ICLR*.

[28] A. Kozlov and D. Koller. 1997. Nonuniform dynamic discretization in hybrid networks. In *UAI*.

[29] A. Kucukelbir, R. Ranganath, A. Gelman, and D. Blei. 2015. Automatic variational inference in Stan. In *NIPS*.

[30] B. Li, M. Grechanik, and D. Poshyvanyk. 2014. Sanitizing and minimizing databases for software application test outsourcing. In *ICST*.

[31] K. Luckow, C. Păsăreanu, M. Dwyer, A. Filieri, and W. Visser. 2014. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *ASE*.

[32] V. Mansinghka, D. Selsam, and Y. Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint 1404.0099* (2014).

[33] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. 2013. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *Journal of Computer Security* (2013).

[34] T. Mens and T. Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* (2004).

[35] T. Minka, J. M. Winn, J. P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2013. Infer.NET 2.5. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[36] G. Misherghi and Z. Su. 2006. HDD: hierarchical delta debugging. In *ICSE*.

[37] D. Monniaux. 2000. Abstract interpretation of probabilistic semantics. In *SAS*.

[38] C. Nandi, D. Grossman, A. Sampson, T. Mytkowicz, and K. McKinley. 2017. Debugging probabilistic programs. In *MAPL*.

[39] R. Neal et al. 2011. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo* (2011).

[40] V. Ngo and A. Legay. 2018. Formal verification of probabilistic SystemC models with statistical model checking. *Journal of Software: Evolution and Process* (2018).

[41] J. Nocedal and S. Wright. 2006. Numerical optimization, series in operations research and financial engineering. *Springer* (2006).

[42] A. Nori, C. Hur, S. Rajamani, and S. Samuel. 2014. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI*.

[43] T. Parr. 2013. *The definitive ANTLR 4 reference.* Pragmatic Bookshelf.

[44] A. Pfeffer. 2001. IBAL: a probabilistic rational programming language. In *IJCAI*.

[45] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. 2012. Test-case reduction for C compiler bugs. In *PLDI*.

[46] RStanIssue 2018. Stan Issue Reporting Instruction. http://mc-stan.org/users/issues/index.html.

[47] D. Selsam, P. Liang, and D. Dill. 2017. Developing Bug-Free Machine Learning Systems With Formal Mathematics. *arXiv preprint arXiv:1706.08605* (2017).

[48] StanExampleModels 2018. https://github.com/stan-dev/example-models.

[49] StanIssue1610 2015. Stan Issue #1610. https://github.com/stan-dev/stan/issues/1610.

[50] StormAppendix 2019. Storm Appendix. https://www.probfuzz.com/storm/appendix.pdf.

[51] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su. 2018. Perses: Syntax-Guided Program Reduction. In *ICSE*.

[52] D. Tran, A. Kucukelbir, A. Dieng, M. Rudolph, D. Liang, and D. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv* (2016).

[53] A. Vahabzadeh, A. Stocco, and A. Mesbah. 2018. Fine-Grained Test Minimization. In *ICSE*.

[54] M. Weiser. 1981. Program slicing. In *ICSE*.

[55] F. Wood, J. van de Meent, and V. Mansinghka. 2014. A new approach to probabilistic programming inference. In *AiStats*.

[56] X. Yang, Y. Chen, E. Eide, and J. Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*.

[57] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering* (2002).

[58] S. Zhang. 2013. Practical semantic test simplification. In *ICSE*.